homework_2

November 5, 2018

```
In [ ]:
```

1 Problem 1- small data learning with embeddings (30%)

1.0.1 We're going to use pre-trained embeddings to try to learn a text classification problem with few training examples

1.0.2 This is very similar to what we did in class!

1.1 $ \ $

1.2 Part 0: Load the data

```
In [1]: # Restart here
```

```
In [1]: import numpy as np
        import pandas as pd
        %pylab inline

        from sklearn.datasets import fetch_20newsgroups
        from keras.preprocessing.text import Tokenizer
        from keras.preprocessing.sequence import pad_sequences
        from keras.utils import to_categorical


        from sklearn.metrics import accuracy_score
```

```
Populating the interactive namespace from numpy and matplotlib


Using TensorFlow backend.
```

```
In [2]: data_train = fetch_20newsgroups(subset='train', remove=('headers', 'footers', 'quotes'
        data_test = fetch_20newsgroups(subset='test',remove=('headers', 'footers', 'quotes'), 
```

```
In [ ]:
```

1.3 Part 1:

1.3.1 a. What is the most common class in the train set?

1.3.2 b. What is the out of sample (test) accuracy if we guess the most probable class?

In [ ]:
```
# hint: you can do this in many ways, including collections.Counter or pandas
print('most common class {}: {}'.format(most_common_class, data_train.target_names[mos
```

In [ ]:
```
# find the accuracy score
```

In [ ]:

1.4 Part 2: Turn the text into integer sequences

In [3]:
```
MAX_WORDS = 10000
MAX_SEQ_LEN = 100
EMBEDDING_DIM = 50
```

In [4]:
```
# TODO
# 1. Instantiate a tokenizer with max workds
# 2. fit the tokenizer on text
# 3. Turn the text into integer sequences (train and test)
# 4. pad the sequences to a constant sequence length (train and test)
# 5. turn y into categorical variables


# you should have 4 variables:
# y_train, y_test, int_sequences_train, int_sequences_test
# all are numpy arrays
```

In [7]:
```
assert y_train.shape ==(11314, 20), 'something went wrong'
assert int_sequences_test.shape == (7532, 100), 'something went wrong'
```

In [ ]:

1.5 Part 3: load the GloVe embedding file

In [ ]:
```
GLOVE_DIR = ''  # FIXME directory with glove
GLOVE_PATH = os.path.join(GLOVE_DIR, 'glove.6B.50d.txt')
```

In [12]:
```
def load_glove_file(filepath):
    word_to_vector = {}
    with open(filepath) as f:
        for line in f:
            values = line.split()
            word = values[0]
            vector = np.asarray(values[1:], dtype='float32')
            word_to_vector[word] = vector
    return word_to_vector
```

```
word_vecs = load_glove_file(GLOVE_PATH)


embedding_matrix = np.zeros((MAX_WORDS, EMBEDDING_DIM))
for word, i in tok.word_index.items():
    if i >= MAX_WORDS:
        continue
    embedding_vector = word_vecs.get(word)
    if embedding_vector is not None:
        # words not found in embedding index will be all-zeros.
        embedding_matrix[i] = embedding_vector
```

In [15]:
```
NUM_CLASSES = y_train.shape[1]
assert NUM_CLASSES == 20, 'something went wrong'
NUM_CLASSES
```

Out[15]: 20

In [ ]:

2 Part 4: Train a model

In [16]:
```
from keras.models import Model
from keras.layers import Input, Embedding, Dropout, Dense, GlobalAveragePooling1D
from keras.initializers import Constant
import keras.backend as K


# TODO
# 1. Build a model with
#   - an embedding
#   - some number of dense layers
#   - dropout
#   - don't forget to use GlobalAveragePooling to average over one dimension


K.clear_session()

word_input = Input(shape=(MAX_SEQ_LEN,), dtype='int32')

# Add code here

# output = ...

model = Model(word_input, output)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

In [17]:
```
model.count_params()
```

Out[17]: 516084

In [18]:
```
num_samples_to_train = 100
epochs = 1000  # this is a big number but won't take long with 100 samples
```

In [19]:
```
model.fit(
    int_sequences_train[:num_samples_to_train],
    y_train[:num_samples_to_train],
    epochs=1000, shuffle=True, batch_size=num_samples_to_train, verbose=0
)
accuracy_score(np.argmax(y_test, axis=1), np.argmax(model.predict(int_sequences_test)
```

Out[19]: 0.2529208709506107

2.1 you should be able to get more than 20% accuracy

In [ ]:

2.2 Part 5: Compare to others methods

2.2.1 a. How does this compare to a randomly initialized, trainable embedding?

In [20]:
```
# TODO
# 1. Build the same model as above, but with a random embedding

K.clear_session()

word_input = Input(shape=(MAX_SEQ_LEN,), dtype='int32')

# your code here

# output = ...

model = Model(word_input, output)
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accurac
```

In [21]:
```
model.fit(
    int_sequences_train[:num_samples_to_train],
    y_train[:num_samples_to_train],
    epochs=1000, shuffle=True, batch_size=num_samples_to_train, verbose=0
)
accuracy_score(np.argmax(y_test, axis=1), np.argmax(model.predict(int_sequences_test)
```

Out[21]: 0.10050451407328731

In [ ]:

2.3 5b: how does this compare to logistic regression trained on 100 samples?

In [22]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import CountVectorizer

data_train = fetch_20newsgroups(subset='train', remove=('headers', 'footers', 'quotes
data_test = fetch_20newsgroups(subset='test',remove=('headers', 'footers', 'quotes'))

# TODO
# 1. make a count vectorizer
# 2. fit it on only `samples_to_train` data points
# 3. trainsform train and test data into integers
# 4. fit logistic regression on just `num_samples_to_train` samples
# 5. Compute accuracy score

vec = CountVectorizer()

# your code here

accuracy_score(data_test.target, lr.predict(x_test))
```

Out[22]: 0.12413701540095592

2.4 This should be approximately 10-12%

In [ ]:

In [ ]:

3 Problem 2: Homework problem: improving BOW (30%)

There are many improvements that can be made to the bag of words represetation, without resorting to neural networks. Here we'll try one

In [35]:
```
# safe to restart notebook
```

In [1]:
```
import numpy as np
import pandas as pd
%pylab inline

from sklearn.linear_model import LogisticRegression
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics import accuracy_score
```

```
Populating the interactive namespace from numpy and matplotlib
```

In [2]:
```
np.random.seed(1234)
```

3.1 Part 1: fit a bag of words and logistic regression to the 20 newsgroups data

In [3]:
```
data_train = fetch_20newsgroups(subset='train', remove=('headers', 'footers', 'quotes'
data_test = fetch_20newsgroups(subset='test',remove=('headers', 'footers', 'quotes'))
```

In [4]:
```
# Todo
# 1. make a count vectorizer with max_features=20000
# 2. fit it
# 3. transform the train and test data into number
vec = CountVectorizer(max_features=20000)
vec.fit(data_train.data)
xtr = vec.transform(data_train.data)
xte = vec.transform(data_test.data)
```

In [5]:
```
# TODO
# 1. fit logistic regression
# 2. compute accuracy score
lr = LogisticRegression()
lr.fit(xtr, data_train.target)   # to be removed
accuracy_score(data_test.target, lr.predict(xte))
```

Out[5]:
```
0.6064790228359002
```

In [ ]:

In [ ]:

3.2 Part 2: TFIDF

A big problem with counting words is that we'll tend to overweight very common words. These common words often carry little information

In [8]:
```
from collections import Counter
def word_iterator():
    """This iterator yields one word at a time from the train data"""
    for doc in data_train.data:
        for word in doc.split():
            yield word

Counter(word_iterator()).most_common(10)
```

Out[8]:
```
[('the', 93969),
 ('to', 51191),
 ('of', 45608),
 ('a', 40042),
 ('and', 39197),
 ('is', 28204),
 ('in', 27756),
 ('I', 27143),
 ('that', 25016),
 ('for', 18066)]
```

3.2.1 TFIDF is a scheme that combats this.

3.3 TFIDF = $Term Frequency Inverse Document Frequency $

4 $ \ $

5 $TFIDF\left(d,t\right) \equiv \frac{\textbf{Count}(d,t)}{\textbf{Doc-Freq}(d,t)} \equiv$ **Count** $\left(d,t\right)\left(1 + log\left(\frac{N_{docs}}{df_t}\right)\right)$

5.1 Where

5.1.1 df_t is the number of documents in which term t appears

5.1.2 N_{docs} is the total number of documents

5.1.3 Count (d,t) is the number of times term t appears in document d (the count matrix)

6 $ \ $

7 $ \ $

7.1 Like this, we suppress the weight of common words

In [ ]:

7.2 2a: write turn the count matrix into a TFIDF matrix

In [5]:
```
def get_idf_vector(count_matrix):
    """Get the inverse document frequence vector (shape = num_words)"""
    df = np.array((count_matrix > 0).astype(int).sum(axis=0))
    return np.log(count_matrix.shape[0] / (1+df))


#TODO(fill in this function)
def get_tfidf_matrix(count_matrix):
    """Turn a count matrix into a tfidf matrix"""
    # TODO
#      1. get the idfs with the aboce function
#      2. turn it into a numpy array `with .toarray()`
#      3. loop the the ROWS of the matrix and transform them
        # YOUR CODE HERE
```

In [6]:
```
xtr_transformed = get_tfidf_matrix(xtr)
xte_transformed = get_tfidf_matrix(xte)
```

In [7]:
```
assert xtr_transformed.shape == xtr.shape, 'something has gone wrong'
print('It worked!')
```

```
It worked!
```

In [8]:
```
lr = LogisticRegression()
lr.fit(xtr_transformed, data_train.target)
accuracy_score(data_test.target, lr.predict(xte_transformed))
```

Out[8]: 0.6251991502920871

In [ ]:

7.3 Part 3: Do the same with scikitlearn's implmenetation

7.3.1 Happily, sklearn does this for us

7.3.2 And it includes some other nice normalization

In [24]:
```
from sklearn.feature_extraction.text import TfidfVectorizer

# TODO:
# 1. instantiate a TfidfVectorizer with max_features = 20000
# 2. fit it on the train data
# 3. transform train and test data into matrices
# 4. fit logistic regression on the train data
# 5. compute the accuracy score on the test data


vec = TfidfVectorizer(max_features=20000)

# Your code here

accuracy_score(data_test.target, lr.predict(x_test))
```

Out[24]: 0.6715347849176846

In [ ]:

7.4 Part 4: Tuning the number of words to use

7.5 Make a plot of how the vocabulary size (max_features) impacts results

In [32]:
```
import time
start_time = time.time()
results = {}
for max_features in (100, 500, 1000, 5000, 10000, 20000, 50000, None):
    # TODO:
    # 1. instantiate a TfidfVectorizer with max_features = max_features
    # 2. fit it on the train data
    # 3. transform train and test data into matrices
    # 4. fit logistic regression on the train data
    # 5. compute the accuracy score on the test data

    # your code here
```

```
        # x_train = ...
        # x_test = ...

        lr = LogisticRegression()
        # lr.fit(...

        if max_features is None:
            num_features = len(vec.get_feature_names())
        else:
            num_features = max_features
        results[num_features] = accuracy_score(data_test.target, lr.predict(x_test)) # to

    print('this took {:.2f} seconds'.format(time.time() - start_time))
```

```
this took 59.59 seconds
```

```
In [ ]: pd.Series(results).plot(figsize=(12,8), fontsize=16)
        plt.xlabel('max features', fontsize=16)
        plt.ylabel('out of sample accuracy', fontsize=16)
```

```
In [ ]:
```

```
In [ ]:
```

8 Problem 3: Named entity recognition (40 %)

Named entity recognition is a common NLP task that tries to identify entities in text.

See: https://en.wikipedia.org/wiki/Named-entity_recognition

Common Types of entities include `Locations`, `People`, and `Organizations`. For example, in the sentence # Janet Yellen, the chairwoman of the Federal Reserve, gave a speech in Colorado. ## $ \ $ the goal would be to recognize # `Janet Yellen`$_{PERSON}$, the chairwoman of the `Federal Reserve`$_{ORGANIZATION}$, gave a speech in `Colorado`$_{LOCATION}$. # $ \ $ # In this problem we will build a model to recognized named entities using word vectors

```
In [ ]:
```

8.1 Part 1:

8.1.1 Give an example of a sentence with a Person but not a location.

8.1.2 Give an example of a sentence with an organization and a location, but not a person.

```
In [ ]: # put it here!
```

9 Part 2: building a model

10 $ \ $

10.0.1 The goal of this section is to build a model to take a sentence (list of words) and identify what kind of tag each word is

11 $ \ $

11.1 Why is this problem hard:

11.1.1 Some words will be the same tag all the time. For example `Colorado` is almost always a `LOCATION`

11.1.2 Some words depend on context: above `federal` and `reserve` are `ORGANIZATION` but I can write `I would like to reserve a table.`

12 $ \ $

12.1 To combat this issue we will make a very simple model but taking a 3-word window around every word

- For every word, we will take the word vector of that word and the two surrounding words

12.1.1 Example: `I went to the store` will be represented as

- `I` $\rightarrow$ `UKNOWN` - `I` - `went` $\rightarrow [V_{UKNOWN}, V_I, V_{went}]$
- `went` $\rightarrow$ `I` - `went` - `to` $\rightarrow [V_I, V_{went}, V_{to}]$
- `to` $\rightarrow$ `went` - `to` - `the` $\rightarrow [V_{went}, V_{to}, V_{the}]$
- ... #### Where
- V_{word_i} is the representation for $word_i$
- `UKNOWN` is the token for unknown or boundary words

Like this, we will encode some **context** around every word. Each word here will be encoded as a $3 * d_{embedding}$-dimensional vector.

In [ ]:

In [5]:
```
import numpy as np
import pandas as pd
%pylab inline

import re

from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, f1_score

from keras.models import Model
from keras.layers import Input, Dropout, Dense
from keras.initializers import Constant
```

```
import keras.backend as K
from keras.utils import to_categorical
```

```
Populating the interactive namespace from numpy and matplotlib
```

```
In [6]: def load_glove_file(filepath):
            """Load a glove embedding from a file"""
            word_to_vector = {}
            with open(filepath) as f:
                for line in f:
                    values = line.split()
                    word = values[0]
                    vector = np.asarray(values[1:], dtype='float32')
                    word_to_vector[word] = vector
            return word_to_vector

        def load_dataset(fname):
            """Load an NER dataset"""
            docs = []
            with open(fname) as fd:
                cur = []
                for line in fd:
                    line = line.lower()
                    # new sentence on -DOCSTART- or blank line
                    if re.match(r"-DOCSTART-.+".lower(), line) or (len(line.strip()) == 0):
                        if len(cur) > 0:
                            docs.append(cur)
                        cur = []
                    else: # read in tokens
                        cur.append(line.strip().split("\t",1))
                # flush running buffer
                if cur:
                    docs.append(cur)
            return docs
```

```
In [7]: import os
        GLOVE_DIR = ''  # FIXME directory with glove
        DATA_PATH = ''  # where you downloaded the data

        word_vecs = load_glove_file(os.path.join(GLOVEIDR, 'glove.6B.50d.txt'))
        docs = load_dataset(DATA_PATH)
```

```
In [8]: correct_first_doc = [
            ['eu', 'org'],
            ['rejects', 'o'],
            ['german', 'misc'],
            ['call', 'o'],
            ['to', 'o'],
```

```
    ['boycott', 'o'],
    ['british', 'misc'],
    ['lamb', 'o'],
    ['.', 'o'],
]
assert len(word_vecs) == 400000, 'word vectors did not load properly'
assert word_vecs['the'].shape == (50,), 'word vectors did not load properly'
assert len(docs) == 14041, 'something has gone wrong with data loading'
assert docs[0] == correct_first_doc, 'something has gone wrong with data loading'
```

In [ ]:

In [9]:
```
MAX_WORDS = len(word_vecs)  # max number of words to use in the embedding
UNKNOWN = 'UUUNKKK'.lower()  # token for unknown word
UNKNOWN_WORD_INDEX = 0
EMBEDDING_DIM = 50  # dimension of embedding
NULL_TAG = 'o'  # tags that are not a named entity

# Some derived quantities
TAGS = (NULL_TAG, 'loc', 'per', 'org', 'misc')
NUM_TO_TAG = dict(enumerate(TAGS))
TAG_TO_NUM = {tag: num for num, tag in NUM_TO_TAG.items()}

NUM_CLASSES = len(TAGS)
assert NUM_CLASSES == 5, 'somethig has gone wrong'

WINDOW = 1
```

In [10]:
```
word_to_num = {word: idx + 1 for idx, word in enumerate(word_vecs.keys())}
num_to_word = {num: word for word, num in word_to_num.items()}

word_to_num[UNKNOWN] = UNKNOWN_WORD_INDEX
num_to_word[UNKNOWN_WORD_INDEX] = UNKNOWN

assert word_to_num['the'] < 10, '"the" is not a common word- something has gone wrong
```

In [ ]:

In [ ]:

In [11]:
```
# Create an embedding matrix
embedding_matrix = np.zeros((MAX_WORDS, EMBEDDING_DIM))
for word, i in word_to_num.items():#tok.word_index.items():
    if i >= MAX_WORDS:
        continue
    embedding_vector = word_vecs.get(word)
    if embedding_vector is not None:
        # words not found in embedding index will be all-zeros.
        embedding_matrix[i] = embedding_vector
```

12.2 Creating windowed-word sequences

In [12]:
```
def seq_to_windows(words, tags, word_to_num, tag_to_num, left=WINDOW, right=WINDOW):
    """Turn sequences of words and tags into corresponding windowed sequences"""
    X = []
    y = []
    word_dict = {ind: word for ind, word in enumerate(words)}
    for i, word in enumerate(words):
        if word == "<s>" or word == "</s>":
            continue # skip sentence delimiters
        word_seq = [word_dict.get(i + ii, UNKNOWN) for ii in range(-left, 1 + right)]
        int_seq = [word_to_num.get(w, UNKNOWN_WORD_INDEX) for w in word_seq]
        tagn = tag_to_num[tags[i]]
        X.append(int_seq)
        y.append(tagn)
    return array(X), array(y)


def window_row_to_vector(window_row, embed_matrix):
    """Turn a row of integers (np.array) into a single word vector"""
    # TODO: implement this
    return np.hstack([embed_matrix[i] for i in window_row]) # to be removed
```

In [ ]:

In [13]:
```
words, tags = zip(*docs[0])
x, y = seq_to_windows(words, tags, word_to_num=word_to_num, tag_to_num=TAG_TO_NUM)
```

In [14]:
```
assert x.dtype == np.int, 'x has the wrong data type'
reconstructed_words = [num_to_word[num] for num in x[:, WINDOW]]
assert tuple(reconstructed_words) == words, 'word transformation has gone wrong'
```

In [ ]:

In [19]:
```
all_xs = []
all_ys = []
for doc in docs:
#     TODO
#     1. unpack the words and the tags from `docs`
#     2. use `seq_to_windows` to turn `words` and `tag` into `x` and `y`
#     3. turn `x` into a single vector with `window_row_to_vector`

    words, tags = zip(*doc)

    # Your code here

    all_xs.extend(x)
    all_ys.extend(y)
```

```
all_xs = np.vstack(all_xs)
all_ys = np.vstack(all_ys)


all_ys = to_categorical(all_ys)
assert all_xs.shape[0] == all_ys.shape[0]
```

In [ ]:

In [21]:
```
# TODO
# 1. make an array of indices to be shuffled with `np.arange`
# 2. shuffle the indices randomly
# 3. use the shuffled indices to shuffle `all_xs` and `all_ys`

# YOUR CODE HERE
# inds = np.arange(....

cut = int(0.8 * all_xs.shape[0])
x_train, x_val = all_xs[:cut], all_xs[cut:]
y_train, y_val = all_ys[:cut], all_ys[cut:]
```

In [ ]:

In [22]:
```
K.clear_session()

#TODO
# 1. build a network with
#   - an input layer
#   - some number of dense layers and dropout

word_input = Input(shape=(x_train.shape[1],))  # to be removed

# YOUR CODE HERE

#output = ...

model = Model(word_input, output)
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy
```

In [ ]:

In [ ]:
```
model.fit(x_train, y_train, validation_data=(x_val, y_val), shuffle=True, epochs=16, ba
```

In [ ]:

12.3 We need to process new sentences so that they can be processed by our network

In [24]:
```
def preprocess_sentence(sentence):
    """Preprocess a sentence into word vectors for the model
```

```
        TODO:
            1. split sentence into words (and make lowercase)
            2. make each word into a 3-word window (use seq_to_windows)
                - this will require the creating some fake tags in the right format
                    in order to pass to `seq_to_windows`
            3. turn each row (3-word window) into a single vector (use window_row_to_vect
            4. turn the list of 150-d vectors into a numpy matrix shape (n_words x 150)
        """
        words = sentence.lower().split()
        # your code here
        #fake_tags = ... we don't know the tags
        # your code here
        #return ...
```

In [25]:
```
assert (
    preprocess_sentence('This sentence has five words').shape ==
    (5, EMBEDDING_DIM * (1 + 2 * WINDOW))
), '`preprocess_sentence` does not work'
```

In [27]:
```
# TODO
# 1. Come up with a sentence
# 2. preprocess it for consumption by the network with `preprocess_sentence`
# 3. use the model to make predictions
# 4. Turn the predicted probabilities into predicted labels
# 5. print the output nicely  (done for you)


# Make up some of your own sentences
sentence = 'Eric Schmidt quit after Netflix Inc announced it would acquire Google Inc

processed_sentence = preprocess_sentence(sentence)
#predictions = model.predict...
#predicted_labels = ...

maxlen = max(map(len, sentence.split()))
for word, label in zip(sentence.split(), predicted_labels):
    print('{} :  {}'.format(word.ljust(maxlen), label))
```

```
Eric      :  per
Schmidt   :  per
quit      :  o
after     :  o
Netflix   :  org
Inc       :  org
announced :  o
it        :  o
would     :  o
```

```
acquire   :  o
Google    :  org
Inc       :  org
for       :  o
17        :  o
dollars   :  o
.         :  o
```

In [ ]:

In [ ]: